Addressing Dynamic Issues of Program Model Checking

Flavio Lerda and Willem Visser

RIACS/NASA M/S 269-2
Ames Research Center
Moffett Field, CA 94035-1000
USA
{flerda, wvisser}@riacs.edu

Abstract. Model checking real programs has recently become an active research area. Programs however exhibit two characteristics that make model checking difficult: the complexity of their state and the dynamic nature of many programs. Here we address both these issues within the context of the Java PathFinder (JPF) model checker. Firstly, we will show how the state of a Java program can be encoded efficiently and how this encoding can be exploited to improve model checking. Next we show how to use symmetry reductions to alleviate some of the problems introduced by the dynamic nature of Java programs. Lastly, we show how distributed model checking of a dynamic program can be achieved, and furthermore, how dynamic partitions of the state space can improve model checking. We support all our findings with results from applying these techniques within the JPF model checker.

1 Introduction

Software is playing an increasingly important role in our everyday lives, but sadly, so does software failure. At NASA this point was made painfully clear in 1999 when the Mars Polar Lander was lost due to a software related problem (estimated cost was $165 million) [Spa00]. Although most agree that many software failures can, and must, be caught during the design phase, it is however often the case that a design phase is either missing by itself, or the tools and techniques to analyze the designs are missing. Hence, testing an implementation is still the number one way of finding errors in software systems. Testing, however, can be very expensive, but more importantly, it is often incapable of finding subtle errors - e.g. timing errors in a concurrent system.

Model checking has been used extensively to find subtle errors in hardware and protocol designs [BLPV95,CW96,Hol91]. However, until recently, model checking has been deemed inadequate to analyze software code, due to the high level of detail often found in code. Now there are many groups, from both industry and academia, that are analyzing source code by model checking. Many of these source code model checkers are based on a translation from source code to the input notation of a model checker: Bandera [CDH$^+$00], Java PathFinder 1 [HP98], JCAT [DIS99] are Java model checkers, and, AX [Hol00] and SLAM [BR00] are C model checkers. A drawback of the translation approach is that certain language constructs are difficult to translate and hence two of these tools, JCAT (dSPIN [IS99]) and AX have extended their back-end model checker (SPIN in both cases [Hol97a]) to improve efficiency.

We adopted a different approach by creating a custom-made model checker for Java. We call this tool Java PathFinder 2, henceforth referred to as JPF. JPF is an explicit state model checker that takes as input Java bytecode. It is structured as a search algorithm that uses a special Java virtual machine (JVM^{JPF}) to execute the bytecode instructions one at a time. In order to implement a depth first algorithm the JVM^{JPF}needs also to have a backtracking capability. The tool itself is written is Java and it's executed by the Java virtual machine (just JVM from now on). By executing the bytecode we can not only analyze all of Java, but we can also analyze programs without source code (e.g. libraries and code down-loaded over the web), and other languages for which bytecode translations exist [BKR98,Taf96,CD98]. Recently JPF has been integrated with the Bandera system [CDH$^+$00]: in this case Bandera doesn't need to do any translation because our tool is able to handle Java directly, but Bandera's functionality of slicing and abstraction are available to improve the model checking.

If one looks at the history of model checking input notations, then it is clear that there has been an evolution from simple guarded command style notations, to ones where more complex data-structures are used. We believe this trend will continue and soon complex dynamic data-structures as well as other features from typical programming languages will be common place. The purpose of this paper will be to highlight some of the difficulties and possible solutions we have encountered in developing an efficient model checker that can handle dynamically evolving software systems. We hope this will help others when developing similar systems.

Although it is clear that static analysis of a system before model checking can greatly benefit the verification, e.g. slicing a system with respect to a certain property to be checked, or finding independent statements to allow partial-order reductions, here we will focus mostly on purely dynamic optimizations for which no prior information is required. The interested reader is referred to [VHBP00] where we discuss static analysis for partial-order reductions and other techniques, such as abstraction, that JPF employs before doing model checking.

Model checking software is often considered hard due to the complexity of the state of the system (this is the premise of state-less model checking [God97, Sto00]). We address this problem in section 2, by first showing how a "large" state can be collapsed to a smaller one, how this can be exploited to improve explicit-state model checking, how a novel form of symmetry reductions on the state can reduce the size of the state space, and lastly, how garbage collection improves model checking. The state-space explosion problem can be reduced, but it almost never goes away. Hence, the more memory one has the larger the programs that can be checked. In section 3 we extend the distributed model checking algorithm first used for SPIN [LS99], that exploits the memory of a number of workstations, to work in the dynamic context of Java. Section 4 contains conclusions and directions for future work.

2 Complexity of the State

One of the first issues we had to address was the complexity of the state. A limitation of the current model checking tools is that they cannot handle dynamic structures. In fact dSPIN [IS99], an extension of the model checking SPIN [Hol97a] used as a back-end in the Java model checker tool JCAT [DIS99], introduces direct support for dynamic allocation.

2.1 The Representation of the State

In creating our own model checker, we were free to choose the representation of the state. Our aim was to be able to handle dynamic allocation efficiently and maintain our representation as close as possible to the one suggested by the programming language. The state is composed of three main components:

static area: is an array of entries, one for each class loaded. Each entry contains the values of the static fields of the class and the monitor associated with it. The monitor contains information on the lock for the class: which thread is holding the lock, which threads are waiting for the lock etc. When a new class needs to be loaded a new entry in the static area is created and its fields and monitor are initialized. Once loaded a class will never be unloaded during the execution of an instruction – but it can be unloaded by a backtracking step.

dynamic area: is an array of entries, one for each object. Each entry contains the values of the fields and the monitor[1] associated with it. Objects are created explicitly by specific bytecode instructions. When an object is created an entry is added in the dynamic area and its fields and monitor are initialized. Objects are not destroyed explicitly in Java, but they can be removed if not referenced anymore (see Section 2.5).

thread list: is a list containing the information relative to each thread. It contains the status of the thread together with other information used by the scheduler, and the stack frames created by the method calls. A new entry is created when a new thread is created, and modified each time the execution of a bytecode instruction changes the state of the thread or one of its stack frames.

These three components are dynamic and they can grow and shrink freely during the execution of the program, not imposing any limit on the size of the state. This is a novel feature, since in both SPIN, where process are allocated dynamically, and dSPIN, where also data can be allocated dynamically, there is still a limit imposed on the size of each state.

[1] The fields and monitor structures are the same used in the static area.

2.2 Collapsing the State

The dynamic features of the Java language, namely, class loading, object creation and method invocation, require a complex data-structure to record the state of the system (see for example our state structure in the previous section). Furthermore, in order to do efficient explicit-state model checking one needs to record the states that have been visited (often using a hash-table). From the examples[2] in Table 1 one can clearly see that it is very inefficient to store the states in their original complex form: for both relatively simple Java programs more than 2kB/state are required. Unlike in a tool such as SPIN where state compression is an option, it is clear that for systems that require a more complex state description, compression should be a requirement.

The "collapse" algorithm has been very successful for state compression in SPIN [Hol97b] and hence we decided to extend it for use within JPF. The rationale behind the collapse method is that when a new state is generated large parts of the state are unchanged. This would seem to call for the state to be stored as the difference from the predecessor, but since states need to be compared to determine if a state has been visited before, this would be inefficient. What the collapse does is to associate to a particular part of the state an index. The state can then be collapsed to a list of indexes indicating which components compose the state itself. The decomposition must be unique so that by comparing the indexes it is possible to determine state equality.

In order to generate the indexes we created a set of pools. Each pool is an ordered set without repetitions. Every time a state needs to be stored it is first collapsed: each component is inserted into a pool, the pool returns the index that corresponds to the position of that component in the pool. If the element was not present in the pool it is added at the end, otherwise the index of the copy already present in the pool is returned. The assumption is that the size of the pool is small enough because each single component appears the same in many states.

In SPIN there are pools for the following state components: global variables, processes and asynchronous channels. Asynchronous channels has no counterpart in Java, but one can think of global variables and static fields, and SPIN processes and JAVA threads to be similar. This would seem to imply that a good first try for our state compression should include a pool for the static area, the dynamic area and the threads. This, however, would be inefficient, since each of these three components has further structure that can be exploited. For example, an assignment to a field of an object would make the dynamic area and one thread change, and hence create two large new pool entries. If we rather use a pool for each stack frame in each thread, one for each monitor and one for each fields data entry then the above field assignment would only change one stack frame entry (the one for the method with the assignment, while leaving all other frames in the thread to collapse to their old values) and one fields entry. When deciding on which components to compress one should always pick components that would

[2] The examples used in this and the following tables are available from the JPF webpage http://ase.arc.nasa.gov/jpf

not change too often, in order to get maximum benefit. We therefore choose a pool for each of the following: fields data (from both the static and dynamic area), monitor data (again shared between static and dynamic area), method stack frames and lastly one for other thread information (such as the thread status, that seldom changes).

As can be seen in Table 1, when the compression algorithm is used the number of different elements in the pools is quite small and the reduction of the memory requirements is impressive. Note that the execution time is reduced as well. When a new state is stored it is compared to other states to see if it has already been visited. This operation is highly inefficient when two uncompressed states are compared because of the complexity of the states themselves, but it is quite efficient when the compressed states (namely arrays of integers) are compared.

Table 1. Comparison of JPF using no compression, collapse, and optimized backtrack

	States	**Transitions**	**Pools Entries**
RemoteAgent	66,425	148,825	1,373
	Memory *(MB)*	**Time** *(sec)*	**State Size** *(bytes)*
No Compression	180.79	227.83	2854
Collapse	12.08	138.65	191
Optimized Backtrack	12.08	54.39	191

	States	**Transitions**	**Pools Entries**
BoundedBuffer	105682	275988	583
	Memory *(MB)*	**Time** *(sec)*	**State Size** *(bytes)*
No Compression	504.82	665.90	5009
Collapse	28.17	297.40	445
Optimized Backtrack	28.16	76.82	445

2.3 Optimizing the Backtrack

Although compression made JPF usable, it was clearly still too slow and used too much memory at run-time to handle large examples. Profiling the system revealed that the problem was the way we handled backtracking. Unlike SPIN we decided to store a copy of each state on the depth-first stack for backtracking purposes — SPIN uses a backwards transition to unwind moves when back-tracking, and only if the state change is too large a copy of the previous state is used. The reason for this is that we work on the bytecode level and often one

Java statement[3] can correspond to many bytecode instructions, hence unwinding each bytecode instruction seemed too complicated. The graph in Figure 1 shows how the memory usage before optimizing the backtracking varies during the visit as new states are reached. In the same graph (with a different scale) the depth of the stack is shown. It is evident how the memory usage is strictly related to the depth of the stack because the uncompressed state is stored on the stack.

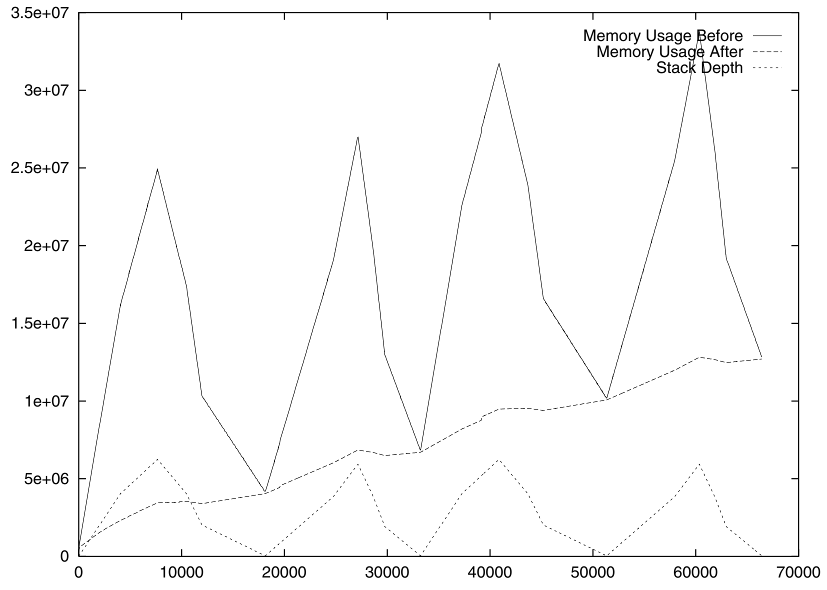

Fig. 1. Memory usage in bytes during execution with and without optimized backtracking. Stack depth is also represented in a different scale

A very simple, and above all novel solution presented itself: store the compressed state on the stack and use the reverse of the collapse operation to recreate the original state when backtracking. In fact, only store a reference to the compressed state on the stack and leave the state itself in the hash-table. Reconstructing the state is quite straight-forward because the collapsed information contains the indexes of the different components of the state that just need to be put back together again via a reverse lookup in the pools (i.e. the original component corresponding to an index must be retrieved). For efficiency only the components that are actually changed are restored. Figure 1 also shows the memory usage after introducing the optimization of the backtrack. A slight

[3] Although JPF executes bytecode instructions, we typically don't use that level of atomicity during model checking, rather we use one JAVA statement or one line of JAVA code as being one transition.

dependency between memory used and stack depth is still present, but now most of the memory is used to store the states. It is interesting to see that the memory usage before and after the optimization intersect where the stack depth comes down to zero – or very close to it.

Table 1 contains the results obtained using this more optimized backtracking technique. It is not evident how the memory usage is optimized (see Figure 1) because the same amount of memory is used when the search is finished. But the execution time is considerably reduced, because it is not necessary to put a copy of the state on the stack anymore.

2.4 Exploiting Symmetries

Symmetries have been used in model checking to reduce the size of the state space [EJ93,ID96,CEJS98,CFJ93,BDH00]. The basic idea is to visit a subset of the state space that is representative of the whole state space based on a symmetry relation that does not influence the properties being checked. Typically symmetry reductions exploit the structure of the system being analyzed, e.g. identical processes, scalar sets etc. [BDH00,ID96]. In keeping with the focus of the paper we exploit symmetries that will be inherent to dynamic systems, and hence we address symmetry issues on the underlying state representation rather than symmetry within the state itself.

```
class A {
  public static int value = 0;
}
```

```
class B {
  public static int value = 1;
}
```

Symmetry	States	Transitions
no	82	152
yes	70	132

```
class Main extends Thread {
  public static void
      main(String[] args) {
    new Main(true).start();
    new Main(false).start();
  }
  public int x;
  private boolean a;
  public Main(boolean a) {
    this.a = a;
  }
  public void run() {
    if(a) {
      x = A.value;
    } else {
      x = B.value;
} } }
```

Fig. 2. Static area symmetry reduction

Specifically we want to exploit the symmetry when classes are loaded into the static area and when objects are created in the dynamic area. Recall from section 2.1 that both the static and dynamic area are implemented as arrays, but the exact position of a class or object in these arrays should not be relevant when comparing states. Nondeterminism, either from concurrency or the environment, can cause classes to be loaded or objects to be created in different orders along different execution paths.

Comparing all possible permutations of the array entries when comparing states is however computationally very expensive, and hence we decided to rather use a canonization function to achieve efficient symmetry reductions. The idea is the following: whenever a state is generated we calculate a canonical representation of the state and use this representation for state comparisons. The use of a canonization function is a well-known technique for achieving symmetry reductions [BDH00,ID96], but calculating this function can be very expensive [CEJS98] in itself. However, since we are only interested in a limited form of symmetry reduction, we can calculate the canonical state representation very efficiently by imposing an order on the entries in the static and dynamic areas. Due to the dynamic nature of Java programs, this ordering must be calculated during model checking. Also, since we use the position of classes and objects as references, an ordering that would require positions to change would be inefficient. The idea is to dynamically map each class (object) to a position in the static (dynamic) array when the class (object) is *first* loaded (created). When backtracking, this mapping is preserved, and reused when executing down a different path.

Figure 2 shows a piece of code where two classes – A and B – are loaded when one of their static fields is accessed. Two threads are executing each one accessing a different class. Depending on the scheduling class A can be loaded before class B or vice-versa. Without symmetry reduction, if classes are allocated in the static area in the order they are loaded, the two interleavings above would lead to two different states, one where A occupies the position before B and the other where B is in front of A. With symmetry reduction we keep a mapping of class names to positions with respect to which class got loaded first, and hence we see a reduction in the number of states (see table in Figure 2). Note that for classes we could have chosen an alphabetical ordering, but this would violate the condition that positions must not change due to the ordering.

Since class names are unique this simple mapping of names to positions is sufficient to achieve a canonical representation for the static area. The dynamic area is not quite as straight-forward, since objects do not have names and hence the different objects created cannot be so easily identified. Our first guess was to use the bytecode instruction that created the object to identify it: this works fine (see results in Figure 3) as long as each instruction is executed at most once during each execution path. However allocation instructions can be executed more than once and create more than one object (e.g. an allocation within a loop). To deal with that we added to the identifier of an object the occurrences of the instruction, i.e. the number of times that instruction has been executed

```
class ThreadOne extends Thread {
  public Object o;
  public void run() {
    o = new Object();
} }
```

```
class ThreadTwo extends Thread {
  public Object o;
  public void run() {
    o = new Object();
} }
```

```
class Main {
  public static void
      main(String[] args) {
    new ThreadOne().start();
    new ThreadTwo().start();
} }
```

Symmetry	States	Transitions
no	54	89
yes	30	51

Fig. 3. Dynamic area symmetry reduction

before. Note that this occurrence number is incremented for an allocation, and also decremented whenever the allocation is backtracked.

```
class ThreadOne extends Thread {
  public Object o;
  public void run() {
    o = Main.newObject();
} }
```

```
class ThreadTwo extends Thread {
  public Object o;
  public void run() {
    o = Main.newObject();
} }
```

```
class Main {
  public static void
      main(String[] args) {
    new ThreadOne().start();
    new ThreadTwo().start();
  }
  public static Object newObject() {
    return new Object();
} }
```

Symmetry	States	Transitions
no	122	237
yes	86	168
thread	68	135

Fig. 4. One instruction can be called by different threads

This is sufficient to identify an object, but it might not lead to the most efficient symmetry reduction results. In Figure 4 the same bytecode instruction is executed by two different threads. The number of occurrences with which each thread executes the instruction depends on the interleaving and therefore so does the position of the object in the dynamic area. One way to address this problem is to add to the identifier of an object the thread that created the objects and count the number of occurrences of the same instruction in each thread separately (see last row in the table in Figure 4).

The same basic approach has been applied to the static and dynamic areas. For the dynamic area – since there is no unique identifier for an object other then the position in the dynamic area itself – it is necessary to use some other information (the bytecode instruction that created the object, the number of occurrences that instruction had before, the thread that created the object) in order to create a dynamic ordering. Although we showed results on small examples in this section, the difference symmetry reductions can make on more realistic examples is quite good: e.g. without symmetry reductions the Bounded-Buffer example from Table 1 has 2502761 states and only 105682 with symmetry reduction (i.e. a 25 fold reduction). Furthermore, the time overhead is almost non-existent: when applying symmetry reduction to an example that have no such reductions we see a 1% increase in the execution time.

The symmetry reduction presented in this paper is orthogonal to other kind of symmetries based on the structure of the system, and therefore they could be used together. Note, that unlike [ID96,BDH00] we don't require an extension of the model checker input language in order to achieve efficient symmetry reductions. Lastly, some of the symmetries that are exploited here are due to different interleavings of some thread execution. Partial order reduction techniques avoid exploring some of those interleavings, and therefore it can mitigate the effect of our symmetry reduction. However our method can reduce symmetries that cannot be avoided by partial order reductions, but more importantly, can be applied without requiring expensive static analysis to determine transition independence.

2.5 Garbage Collection

Java does not offer any primitive to deallocate an object. Once created an object will continue to exist until it is *garbage collected*. An object can be garbage collected when no more references to it are available.

If we want to model check software written in Java we need to take into account garbage collection. Many Java programs rely on its presence, and even very simple examples – see Figure 5 – would have an infinite number of states without garbage collection (each allocation increases the size of the state, hence causing an infinite state-space). Garbage collection during model checking was first introduced in [IS00] and we implemented the same two algorithms, namely reference counting and mark and sweep, in JPF. The results presented in Table 2 show how reference counting and mark and sweep perform in our implementation. Three examples are analyzed:

```
public class Main {
  public static void main(String[] args) {
    Object o;
    while(true) {
      o = new Object();
} } }
```

Fig. 5. Even a simple example can have infinite states

- **TempObj** creates many temporary objects;
- **NoGarbage** adds elements to a list but never removes them, creating no objects to be collected; and
- **DoubleLinked** creates a double linked list and then loses the only pointer to it.

All these examples include concurrency (two or more threads are executing the same operations concurrently). In the table the following have been reported: the number of states, the number of transitions, the memory usage (at the end of the verification), the execution time, and the number of objects collected (with the number of times the algorithm has been activated for mark and sweep).

Table 2. Results obtained with GC

TempObj	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	609173	1276971	168.26	508.24	-(-)
Mark and Sweep	2923	7110	1.1	5.73	2750(4286)
Reference Count	2923	7110	1.1	6.48	2750(-)

NoGarbage	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	833766	1812626	216.88	520.26	-(-)
Mark and Sweep	833766	1812626	216.95	646.43	0(1241203)
Reference Count	833766	1812626	216.95	621.90	0(-)

DoubleLinked	**States**	**Transitions**	**Memory** *(MB)*	**Time** *(sec)*	**GC** *objects(runs)*
no GC	124503	310006	71.31	99.18	-(-)
Mark and Sweep	124503	310006	33.45	101.98	35928(231163)
Reference Count	124503	310006	71.37	120.01	0(-)

The first example – TempObj – is heavily affected by garbage collection. The fact that temporary objects are created – which is quite common in Java – adds extra information to the state that makes equivalent states seem different if garbage collection is not active. The memory requirement are the same for both algorithms, but mark and sweep is slightly faster, since the ratio of objects collected per run is quite high (this is not the case for the other two examples).

The second example – NoGarbage – shows the overhead introduced by the two algorithms, since this example does not produce any object to be collected. The extra memory is about the same for both algorithms, but that's not true for the execution time. The mark and sweet algorithm is activated many times and that is reflected in the higher execution time: almost 70% of the transitions cause the algorithm to be executed. The algorithm is activated only after an instruction that can produce garbage: unfortunately many of the bytecode instructions can.

The last example – DoubleLinked – leads to two considerations. First, the reference count algorithm is not able to detect cycles of garbage. As a consequence no garbage is detected by this algorithm. Secondly, even when garbage is found with the mark and sweep algorithm, the state space is not reduced: the reason is that states with their garbage removed are still different because of other variables in the program. However, there is a reduction in the memory since states from which garbage was collected are now smaller.

Note, unlike the case for dSPIN[IS00], where it was necessary to store the complete state on the stack before garbage collection – because the state was changed in an irreversible way – we do not need to do so because the collapsed version of the state is already stored on the stack.

3 Distributed Memory

Explicit state model checking suffers from the state explosion problem, and when analyzing software programs this problem is more severe due to the higher-level of detail present in such programs. In order to deal with this issue, many different solutions have been tried, distributed model checking being one of them [LS99, SD97]. In this section we present how we improved this technique and adapted it to the dynamic nature of the systems that can be checked with our tool. Our work is based on [LS99], that presents a distributed memory implementation of SPIN. Our goal was to analyze the issues of implementing a distributed model checker when the input model is a dynamic system, in order to guide us in the development of a parallel model checker.

The algorithm presented in [LS99] extends the standard depth-first visit algorithm into a distributed visit algorithm. This new algorithm is no longer depth first but it still visits all the states and paths of the system. The only real issue with this algorithm is that it does not allow LTL model-checking, but this limitation had been overcome in [BBS00].

The basic idea is to divide the state space in partitions. Each node – workstation – will store the states that belong to one of the partitions. Every time a new state is reached a partition function is used to determine which node is the owner of the state and the state is sent there for storage and analysis of the state's successors. If the state has to be visited in the same node then the visit continues depth first from that node. The node the starting state belongs to will start the visit while the others are waiting for incoming messages. The search is completed when all nodes are waiting and all messages sent have been received. The major issue with this algorithm is picking a partition function that minimizes the memory required by each node, but at the same time limits the number of messages required between the nodes.

3.1 Improvements

The algorithm from [LS99] has been adapted to our system, at first without any modification. After making the tool operational we worked on some extensions, mainly aiming to reduce the communication overhead. In [LS99] a modification of the algorithm, called *sibling storing*, is presented, which reduces the number of messages sent. Each time a new state that needs to be sent to another node – sibling – is reached, a local copy of the state sent is kept in the local hash table. If encountered again, no messages need to be sent, since we know that it has been received by that node before. One issue with this technique is that the number of siblings can grow quickly and consume too much of the memory, taking space that could be allocated to store actual (local) states.

We developed a modified version of this technique, that we called *sibling caching*, that stores the siblings in a cache. When no empty space is left in the cache, the least recently used element is discarded and the new sibling is added. This technique proved to be quite effective – see Table 3 – because a very limited size cache performs, in terms of messages avoided, almost as good as complete sibling storing. Table 3 shows the traditional partition algorithm (that uses a hash function over the complete system state for partitioning) augmented with a number of optimizations techniques and compares these with respect to memory usage, percentage of transitions that generates messages, and lastly the time taken.

Table 3. Results using different optimizations with the RemoteAgent example

RemoteAgent	**Memory** *MB*	**Messages** *%*	**Time** *(sec)*
Normal Distributed	40.69	38	525.16
Sibling Storing	45.70	27	504.65
Sibling Caching (50)	40.10	34	518.52
Sibling Caching (200)	39.60	31	515.26
Sibling Caching (500)	39.19	28	511.77
Children Lookahead	29.67	31	320.75
Children Lookahead + Sibling Storing	33.83	23	296.87
Children Lookahead + Sibling Caching (50)	29.19	28	316.11
Children Lookahead + Sibling Caching (200)	29.14	28	314.61
Children Lookahead + Sibling Caching (500)	28.86	26	319.06

Another extension we developed is called *children lookahead*. This technique tries to avoid sending messages due to short paths that fall into another node's state-space. As can be seen in Figure 6, state $s_{i-1}^{(0)}$ is followed by state $s_i^{(1)}$. This generates a message from node 0 to node 1. When node 1 receives the new state it generates its successor $s_{i+1}^{(0)}$. This node needs to be stored by node 0 and so a second message is generated. This last message could have been avoided if node 0 had checked the successors of $s_i^{(1)}$ for any state belonging to itself – this operation being not very expensive because state $s_i^{(1)}$ has already been generated by node 0.

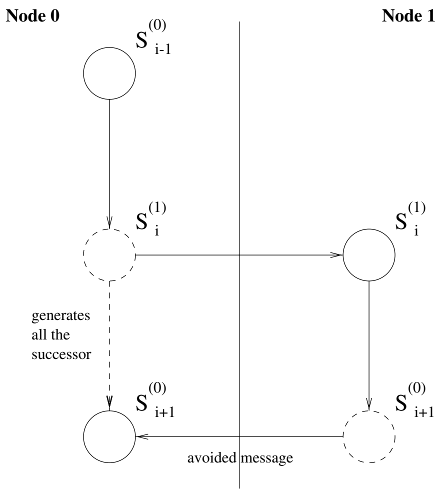

Fig. 6. Example of children lookahead

In order for this technique to work it is necessary to clearly specify who is going to take care of each state. When a message is sent the sender will check for its own states among the successors of the sent state. On the other hand the receiver will skip every state belonging to the sender that falls in the first generation starting from the received state. This algorithm works also if there are more than two parties involved, because the sender will just ignore states belonging to a third party, while the receiver will send the state to the correct node – therefore avoiding duplication. It is important however for the node to check for possible states belonging to itself in the first generation of the state sent to the third node.

This technique avoids messages for runs that last only one state in another node's part of the state space, but the technique can be generalized to an arbitrary number of steps, corresponding to the number of generations that need to be checked. There exists a trade off between the number of generations checked – and therefore the number of possible messages avoided – and the time overhead necessary to generate all the successors – which grows exponentially with respect to the number of generations.

Some results can be seen in Table 3 where the same example has been executed with different combinations of the presented techniques: sibling storing reduces the number of messages more consistently than sibling caching, but increasing the size of the cache – 50, 200, and 500 in the table – the percentage of messages gets closer to the results obtained using storing. Children lookahead appers to be very effective and is orthogonal to the other techniques. The results in terms of message reduction are mirrored by the reduction in execution time. The memory usage should be higher with sibling storing, decrease with the caching and be minimum without any sibling algorithm. Nevertheless the experimental results give an anomalous behavior that we were not able to explain.

State Transfer. A difference between SPIN and JPF is where most of the execution time is spent: in SPIN storing the state uses most of the time, while in JPF execution of the bytecode instructions is the most expensive operation. This is due, in part, to the fact that in JPF transitions are more complex than in SPIN. This difference can affect the design of the distributed version of the two tools. In [LS99] the communication protocol between the nodes had been designed so that a path is sent to identify the state that needs to be visited. This is consistent with the assumption that steps can be executed very efficiently in SPIN. On the other hand, in JPF an execution step is very time consuming so it would sound efficient to send the state, without any need for the path. In JPF however the state is a very complex structure that includes references. At the early stages of the development we tried to send states, but the time necessary to translate the states into something that can be sent on a socket was too high. Therefore our choice was to send the path and use that information to reconstruct the state on the destination node. Although we are sending a path at the moment we believe that when doing the implementation on a parallel architecture sending states would become viable.

Another possibility that we are currently exploring is to send across a compressed version of the state. Since the state is very efficiently compressed by the tool for storing, it would be effective to send the compressed state over the network. This is not at the moment possible because the pools used for the compression are local to each node and therefore it would be impossible to correctly reconstruct the state on the receiving node. One possible approach – that is particularly interesting in a parallel environment, but it's still applicable in a distributed one – would be to centralize the pools used for the compression. This way indexes for components of the state would be global and the compressed state could be easily and quickly transferred between nodes. In order to reduce the communication each node could keep a copy of the entries that it accessed from the centralized pool and only when a new entry has to be added, communication is necessary.

3.2 Partitioning

Partitioning is a crucial point in the distributed algorithm [LS99]. Partitioning aims to achieve two contrasting goals, with an obvious trade off:

– reduce the number of message that need to be transmitted; and
– maintain a fair partitioning of the memory required on each node.

In [LS99] a few heuristics to determine a partition function are suggested. In [Ler00] a more complete approach to the problem is given, and a tool to automatically generate a partition function from static analysis of the input model is presented. However, because of the dynamic nature of the systems we address, these kind of tools are more difficult to implement due to the complexity of the static analysis required. We will therefore focus on partition functions that can be calculated dynamically and compare them to static partitioning functions that do not require any static analysis. In [Ler00] partition functions are classified as:

static: the partitioning is made before the verification is run and no changes are possible once at run-time; or

dynamic: the partition function is adapted at run-time using the information gathered during execution to better suit the system that is being model checked.

These two kinds of partition functions have their advantages and disadvantages: static partitioning does not require further communication to determine which node a state belongs to but it is hard to come up with a good function, i.e. one that achieves both equal partitions and low communication. On the other hand, dynamic partitioning requires a higher level of communication and complexity, but allows more versatility (it is not model dependent) and equal partition size. All the partition functions used in [LS99] are static but the algorithm presented does not rely on that assumption. Static partitioning is especially problematic for a dynamic system, because it is hard to extrapolate what the behavior and structure of the system will be. Therefore a dynamic approach, since it is not dependent on the system structure, seems more appropriate when analyzing dynamic systems.

3.3 Static Partitioning

First we present some examples using static partitioning – Table 4 – that can be used as a reference for the results presented further on. They are also important because – as we will see in Section 3.4 – these partition functions are used as a basis for the dynamic ones.

The results in the table are for the RemoteAgent example using two workstations. The different partition functions are reported in the first column, followed by the percentage of the total memory used by each workstation, the percentage of transitions that cause messages to be sent, and lastly the time taken.

The Global Hash Code partition function uses a hash function to determine the partition: the function is applied to the whole state and the partition is the result modulo the number of partitions. This solution gives a fair division of the state space between the nodes, but at the same time, the number of messages generated is pretty high.

A possible approach is to use the locality principle [LS99]: if the partition function relies only on the information of a particular thread, only when that thread is scheduled is it possible to reach a state that generates a message. As a first step we created a partition function – Local Hash Code – that applies the

Table 4. Different static partition functions on the RemoteAgent example

RemoteAgent	**Memory** *%*	**Messages** *%*	**Time** *(sec)*
Global Hash Code	50/50	38	525.16
Local Hash Code	50/50	46	551.28
Local Hash Code (1)	44/56	11	241.46
Local Hash Code (2)	47/53	13	263.57
Program Counter (1)	54/46	17	342.34
Program Counter (2)	48/52	25	487.43
Program Counters (1)	54/46	17	339.46
Program Counters (2)	43/57	14	326.66

hash function to the thread list only. This implies that the value of the objects are not included in the hashing process – only the stack frames and thread status of all threads. As a further step we limited this process to a specific thread (indicated by a number in the table). The results, in terms of message ratio, are still not very good for the function if applied to the whole thread list because at each step at least one of the program counters will change, but if applied to a single thread, messages are reduced. The choice of the thread is also important: the first thread – zero – is usually a bad choice (hence it was not included in the table), since it is the main thread that often is simply used to create the threads that compose the real system. In general a reduction of the messages is obtained, with a sacrifice in the fairness of the partition.

As said before, in [Ler00] a tool to generate a partition function using static analysis has been presented. Unfortunately, this tool cannot be applied, as is, here because it uses the flow control graph of a thread, that is not as accessible as in SPIN. The idea behind it is to use the current state of a process to determine the partition the state belongs to: on a similar path we tried to use the program counter of threads to do this. Since the static analysis approach cannot be used, we just hashed the program counter. At first sight the program counter seems to be the equivalent of the current state of a Promela process, but because of the stack based approach, each thread has more than one program counter. At first we tried to use the program counter from the topmost stack frame (rows saying Program Counter as partition function), then we tried the same approach using a function of all the program counters from every stack frame (rows marked as Program Counters). The result is a reduction of the percentage of messages, which is higher when the function is applied to a specific thread. Note again, that the main thread (thread zero) is not shown, since as before it is only used to start the rest of the system and hence leads to a very unbalanced partitioning.

An interesting observation is that one can either use too much information to calculate the partitioning, in which case the partition is fair but creates too many messages (see Global Hash Code and Local Hash Code) or one can use too little information (see Program Counter 1 and 2) with similar problems. Using just enough information seemed to give the best results: Local Hash Code for a thread and Program Counters per thread.

3.4 Dynamic Partitioning

The great advantage of dynamic partitioning is that no prior knowledge or static analysis of the system are necessary: run-time information is used to keep the partitioning fair. Dynamic partitioning just means that states can be stored in one node at a certain moment and in another one later on. In general a dynamic partition function will initially assign a subset of the state space to each node and when a certain condition arises – for instance lack of main memory – it will reassign some states to a different node.

In our work we assume that at any given time each node knows where each state is supposed to be stored. This means that a node does not need to interrogate every other node to know if a state has already been visited, but can send it to the legitimate owner. This assumption can be dropped a for limited time after a reassignment with the condition that nodes relay the incoming states that do not belong – anymore – to them to the correct designated node.

One issue that arises at this point is how to represent a partition function that can change with time. It is necessary for some sort of table to specify which state is stored where. It is obvious that the granularity of this table cannot be the single state, otherwise the size of the table would be of the same order of magnitude of the size of the whole state space. States can be grouped together: we called these groups of states *classes*. It is clear that the classification is equivalent to the partitioning. The assumption we made here is that the same techniques used for determining static partition functions can be used to determine a classification function. What is important is that classes do not need to be the exact same size. In fact the number of classes is greater then the number of partitions, and each partition consists of a set of classes: at run-time classes will be grouped together into partitions and when a partition's size is too big, part of it – a class – can be assigned to another node. Still important is to minimize the number of potential messages between two classes, but we do not have a strong trade off like we used to. This solves the problem of having an excellent partition function, since an average classification function can give optimal results.

When a reassignment is issued the states that have already been visited but now belong to a different partition have to be discarded. It is not efficient – at least in a distributed environment – to transfer that information across the network. The node those states are assigned to will rediscover those states – if they will ever be met again – without actually influencing the result of the computation, just extending the search – since some states may be visited more than once.

Table 5 show the results obtained using different dynamic partition functions based on the static ones presented in the previous section. Half of the classes are initially assigned to each node and, if necessary, they will be reassigned. When comparing these results with the static partition results from Table 4 it is clear that in every case the dynamic partition achieves either a similar or better memory distribution and runtime.

An important issue is to decide when a reassignment is necessary: an option would be to start it when the number of states stored in one partition is too

Table 5. Results from dynamic partition with classes split equally

RemoteAgent	**Memory** %	**Messages** %	**Time** *(sec)*
Global Hash Code	50/50	38	524.02
Local Hash Code	50/50	46	531.17
Local Hash Code (1)	48/53	11	216.56
Local Hash Code (2)	48/52	13	281.25
Program Counter (1)	52/48	17	340.97
Program Counter (2)	48/52	25	487.46
Program Counters (1)	52/48	17	311.22
Program Counters (2)	49/51	13	333.78

big compared to what is currently stored in the others. However this is not a good idea, since having a greater amount of states stored in one partition is not necessarily a problem until memory comes to exhaustion. It is better to wait until the memory become an issue than keep the two partition at the same size during the whole visit – also because a class' size can change, for instance if most of its states will be visited close to the end of the verification.

Another issue is that, even when the memory is abundant, the two nodes have to communicate intensively since the beginning, because the states have been divided as equally as possible before starting. One possible optimization would be to store all the states – at least initially – on the same node and let the reassignment and the dynamic algorithm do the work of obtaining a better partitioning. This last approach leaves all but one node completely useless from the beginning up to the time the first node exhausts its memory resources and starts splitting the state space – and the work – with the others. One disadvantage is that a lot of work might need to be redone, because every time a class is reassigned a part of the state space that has already been visited is lost. Table 6 show the results of doing this form of dynamic partitioning — the fairness of the partitioning is still very good, but now the messages and hence the time is much reduced.

Table 6. Results from having one node start with all the classes

RemoteAgent	**Memory**	**Messages** %	**Time** *(sec)*
Global Hash Code	47/53	16	251.74
Local Hash Code	47/54	13	150.09
Local Hash Code (1)	45/55	5	97.57
Local Hash Code (2)	53/47	7	125.09
Program Counter (1)	45/55	11	141.38
Program Counter (2)	44/56	11	134.14
Program Counters (1)	39/61	10	145.38
Program Counters (2)	48/52	14	182.00

One possible way to avoid having idle workstations is to assign to each node a set of states that belong to it, but let them, at the beginning, visit and store also other states. This way all nodes will start visiting the state space at the same time without any need to send messages, because states can be stored in their own hash table. When memory becomes an issue, those states that belong to others can be discarded to make space for local states, but after that messages needs to be sent for those nodes falling in that part of the state space. At first this technique seems very similar to sibling storing but the difference is that states are stored without sending a message. In fact if the successors of a given state are fully visited by one node, the search will be correct even if later on this state will be discarded.

To clarify this technique let's suppose we have only two nodes. Initially both the workstations start the visit until they reach a moment in time when memory becomes scarce. At this point each node will have to discard a part of the state space. Let's assume for simplicity that only two classes were defined: each node will keep one of them and reject the other. With a minimum amount of coordination – to avoid that both reject the same class – both nodes now have only one class stored in their hash table. If we suppose for simplicity again that both nodes got to the exact same point in the visit when they decided to reassign one of their classes, no state would be lost, because each node is keeping what the other rejected – see Figure 7.

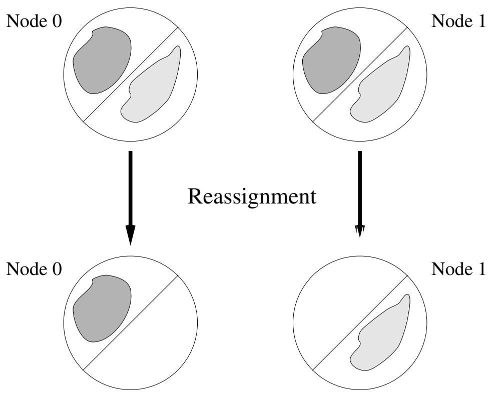

Fig. 7. How states are visited and discarded

This was a simplification: in a more realistic scenario there would be a number of states that are lost, but surely quite limited compared to the same situation where only one node had been running until the first reassignment started. Moreover if more classes than partitions exist the same dynamic techniques applied before can be used. At any time each class can be either stored by a specific node or shared among a set of them. When memory is scarce a shared class can be discarded – making sure ahead of time that not everybody else discard it at the same time – or a non-shared class can be transfered to another node. Table 7 shows results from this form of dynamic partitioning — note how the messages and time is even further reduced from Table 6.

Table 7. Results from having two nodes start together

RemoteAgent	**Memory** %	**Messages** %	**Time** *(sec)*
Global Hash Code	50/50	9	251.17
Local Hash Code	48/52	7	120.92
Local Hash Code (1)	52/48	3	74.46
Local Hash Code (2)	48/52	2	80.65
Program Counter (1)	38/62	4	113.21
Program Counter (2)	49/51	4	122.66
Program Counters (1)	48/52	5	112.46
Program Counters (2)	52/48	3	113.84

4 Conclusions

Program model checking is an area of active research since the importance of software and its failures is increasing. Model checking of software presents specific issues that are due to the complexity and the dynamic nature of programs. Translation-based approaches cannot adequately deal with these since they rely on underlying tools that are not designed to exploit programs specific characteristics. We developed our own model checking tool using a programming language, Java, as our input notation in order to be able to overcome this limitation.

We first introduced a representation for the state that respects the paradigm underlying the input notation. In order to be able to explore the state space of a reasonable size we developed a compression algorithm that exploits the structure of the system state. A novel approach to efficient backtracking has been presented, that reconstructs the state from the compressed version present on the stack. A novel approach to symmetry has been introduced, that exploits symmetries inherent to the state representation. Garbage collection is discussed as a further way to reduce the state space.

Even after applying state space reduction techniques programs are often still too large for the memory of a single workstation: a distributed memory algorithm can overcome this. We show how an existing distributed model checking algorithm can be extended to reduce communication overhead and do dynamic memory balancing. We show results supporting our claim that dynamic partitioning of the state space over multiple workstations is well suited to analyze dynamic (Java) programs. Although we did not show that the dynamic partition functions presented here allow the verification to converge, we believe this is the case. We intend to address this issue formally for more complicated functions that we are currently developing.

This paper focussed on techniques that can be applied without any prior knowledge of system structure. We do however believe that many reduction techniques based on a-priori static analysis of the system, such as slicing, partial order reductions, abstractions, etc., can improve the model checking process and should be applied whenever possible.

In the future, we intend to further investigate the combination of static and dynamic reduction techniques to combat the state explosion. Furthermore, we believe that parallel model checking will become more popular in the future due to the use of such machines becoming more widespread. To this end we are currently extending our distributed model checking algorithm to be used on a parallel shared-memory architecture (SGI Origin 2000).

References

[BBS00] J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. Technical Report FIMU-RS-2000-10, Faculty of Informatics, Masaryk University, 2000. Available in this LNCS volume.

[BDH00] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[BKR98] Nick Benton, Andrew Kennedy, and George Russell. Compiling standard ML to Java bytecodes. *SIGPLAN Notices*, 34(1):129–140, September 1998.

[BLPV95] J. Bormann, J. Lohse, M. Payer, and G. Venzl. Model checking in industrial hardware design. In *Proc. of the 32nd Design Automation Conference*, 1995.

[BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 113–130. Springer-Verlag, September 2000.

[CD98] L.R. Clausen and O. Danvy. Compiling proper tail recursion and first-class continuations: Scheme on the Java Virtual Machine. *The Journal of C Language Translation*, 6(1):20–32, April 1998.

[CDH$^+$00] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and R. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd International Conference on Software Engineering*, June 2000.

[CEJS98] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, and A. Prasad Sistla. Symmetry reductions in model checking. In *Proc. of the 10th International Conference on Computer Aided Verification*, volume 1427 of *LNCS*, pages 147–158. Springer-Verlag, 1998.

[CFJ93] Edmund M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetries in temporal logic model checking. In *Proc. of the 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*. Springer-Verlag, 1993.

[CW96] Edmund M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, 1996.

[DIS99] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[EJ93] E. Emerson and C. Jutla. Symmetry and model checking. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*. Springer-Verlag, 1993.

[God97] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proc of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 476–479. Springer-Verlag, June 1997.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[Hol97a] Gerard J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Hol97b] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of the 3th International SPIN Workshop*, April 1997.

[Hol00] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[HP98] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 1998.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):47–75, August 1996.

[IS99] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.

[IS00] Radu Iosif and Riccardo Sisto. Using garbage collection in model checking. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 20–33. Springer-Verlag, September 2000.

[Ler00] Flavio Lerda. Model checking: Tecniche di verifica formale in ambiente distributo. Master's thesis, Politecnico di Torino, May 2000.

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-memory model checking with SPIN. In *Proc. of the 5th International SPIN Workshop*, volume 1680 of *LNCS*. Springer-Verlag, 1999.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murphi verifier. In *Proc. of the 9th International Conference on Computer Aided Verification*, volume 1254 of *LNCS*, pages 256–278. Springer-Verlag, June 1997.

[Spa00] SpaceViews. Premature engine cutoff likely cause of Mars Polar Lander failure. *http://www.spaceviews.com/2000/03/28b.html*, March 2000.

[Sto00] Scott D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, September 2000.

[Taf96] S. Tucker Taft. Programming the Internet in Ada 95. In *Ada-Europe International Conference on Reliable Software Technologies*, volume 1088 of *LNCS*, pages 1–16. Springer-Verlag, June 1996.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering*, September 2000.